A PDP-8 Emulator Program

BRIAN J. SHELBURNE
Wittenberg University

The clean, simple, and elegant architecture of the classic PDP-8 makes it an ideal candidate for studying concepts in computer organization. The PDP-8 Emulator Program allows a user to write, edit, assemble, debug, trace and execute PDP-8 machine code and PDP-8 assembler language programs. With it the user can obtain a feel for the PDP-8. The PDP-8 Emulator Program includes a simple built-in text editor which is used to write and edit PDP-8 Assembler Language programs, an assembler to translate these programs into PDP-8 machine code and a virtual PDP-8 engine upon which to execute the code. PDP-8 code can be executed from a *debug* screen display that allows the user to observe the contents of registers and memory as the code executes or code can be executed using an I/O interface that requires user-written PDP-8 I/O routines. This paper provides an introduction to both the PDP-8 architecture and PDP-8 Assembler Language and discusses how to use the PDP-8 Emulator Program.

The PDP-8 Emulator Program was written using Borland's Turbo Pascal v 6.0 and runs under MS-DOS in a Command window.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization – General**]: Modeling of Computer Architecture; B.0 [**Hardware**]: General: PDP-8; I.6.5 [**Simulation and Modeling**]: Model Development; K.2 [**History of Computing**]; K.3.1 [**Computers and Education**]: Computer Uses in Education
General Terms: Design
Additional Keywords and Phrases: computer architecture simulator, education

---

1. INTRODUCTION

The PDP-8 Emulator Program (pdp8main.exe) simulates the PDP-8 architecture on an Intel 80x86 computer. With it a user can edit and execute PDP-8 machine code or PDP-8 assembler language programs and thereby develop a feel for the PDP-8. The design philosophy behind the PDP-8 Emulator Program was to incorporate as much as possible the underlying architecture of an actual PDP-8. This philosophy accounts for some of the PDP-8 Emulator Program's unusual features. Much of the structure, terminology and notation used in the PDP-8 Emulator Program reflects that used by a real PDP-8.

The PDP-8 Emulator Program makes use of a number of display screens. One display, the *debug screen*, allows the user to observe the contents of memory and all registers. From this display screen a user can enter PDP-8 machine code directly into memory and observe its execution. Another display is a small built-in text editor that allows the user to create and edit PDP-8 Assembler Language (PAL) programs. An assembler can be invoked to translate the PAL code

Author's address: Department of Mathematics and Computer Science, Wittenberg University, Springfield Ohio, 45501 USA; e-mail bshelburne@wittenberg.edu
Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2001 ACM 1531-4278/01/0600-0001 $5.00

into machine code that can be executed by the virtual PDP-8. PDP-8 programs can be executed from the debug screen or executed under a simple I/O interface called the Run PDP-8 screen. The latter requires writing PDP-8 I/O routines.

Use of the PDP-8 Emulator Program requires some understanding of PDP-8 architecture. The PDP-8 is a 12-bit word addressable machine with a single 12-bit accumulator and 4096 words of memory partitioned into 32 pages of 128 words each. Instructions are fixed length and are 12-bits wide. A three-bit opcode field permits only eight instructions; although two of the eight instructions (opcodes 6 and 7) use the other nine bits as extended opcodes. In particular opcode 7 is a family of orthogonal micro-instructions which can be combined to generate a surprisingly rich set of operations. There are four addressing modes: zero page addressing, current page addressing (both forms of direct addressing), indirect, and auto-index indirect. The first two reflect the partition of memory into 32 pages of 128 words each. The last one is used to handle data structures like strings and arrays.

The PDP-8 Emulator Program only supports the features found in a minimally configured PDP-8. For example it only supports integer addition; other operations like multiplication and division must be done in software. This design decision was made to keep the PDP-8 Emulator Program simple and easy to use yet despite its simplicity the PDP-8 is elegant in its ability to implement complex operations with a limited instruction set. A real PDP-8 had the option of including an Extended Arithmetic Element unit that did multiplication and division in hardware.

The PDP-8 Emulator Program can accept and execute PDP-8 machine code or it can assemble and execute PDP-8 Assembler Language (PAL) programs. Since assembler is the primary way to write programs using the PDP-8 Emulator Program much of the following paper is a quick tutorial in PDP-8 Assembler Language.

Why the PDP-8? The PDP-8 was introduced in 1965 and by today's standards its architecture and capabilities are archaic (truly a *dinosaur* except the PDP-8 was *small*). However its simple design and low cost made the PDP-8 a very successful computer (50,000 units were produced). Because of its simple design PDP-8 assembler is easy to understand, write, and use. Even PDP-8 machine coding can be done without too much difficulty! Assembler languages for today's machines have a steep learning curve and machine code is out of the question. A toy computer simulator could provide an easy assembler (and the capability of easy machine coding) but toy architectures are not real. The PDP-8 Emulator Program simulates a *real* architecture (warts and all) that is simple

and easy to understood making assembly language and machine code programming relatively easy to do - if you want to go this way.

Like the PDP-8, the PDP-8 Emulator Program is old, written almost ten years ago using MS-DOS based Borland Turbo Pascal version 6.0. It has been used since then in my computer organization course as a *gentle* introduction to von-Neumann architectures and assembler language programming. Despite its age it runs well in an MS-DOS Command window.

2. AN OVERVIEW OF THE PDP-8 EMULATOR PROGRAM

2.1 The Main Menu Screen

At the Main (opening) screen the user is presented with four options (plus Quit). To make a selection the user can use arrow keys to highlight the option and hit (Enter) or he/she can type the first letter: D, E, H, R, Q. (Esc) can be used to return to the Main screen or from the Main screen (Esc) can be used to quit the program.

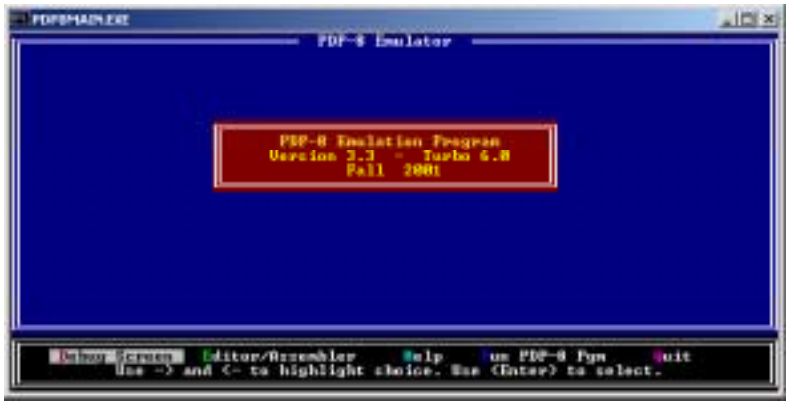


Fig. 1. Main Screen PDP-8 Emulator Program

Debug Screen: This allows the user to directly view the contents of memory and all registers. The user can enter, execute and trace (single-step) simple PDP-8 programs.

Editor/Assembler: A simple text editor allows the user to create, edit, debug, and assemble PPD-8 Assembler Language (pal) programs. A successfully assembled program can be executed in the Debug Screen display or through the Run PDP-8 Pgm Screen display.

Help: On-line help is provided for the PDP-8 Assembler Language and for the PDP-8 Emulator Program. [F1] also invokes help.

Run PDP-8 Program: This provides a simple I/O interface for executing PDP-8 programs.

2.2 The Debug Screen

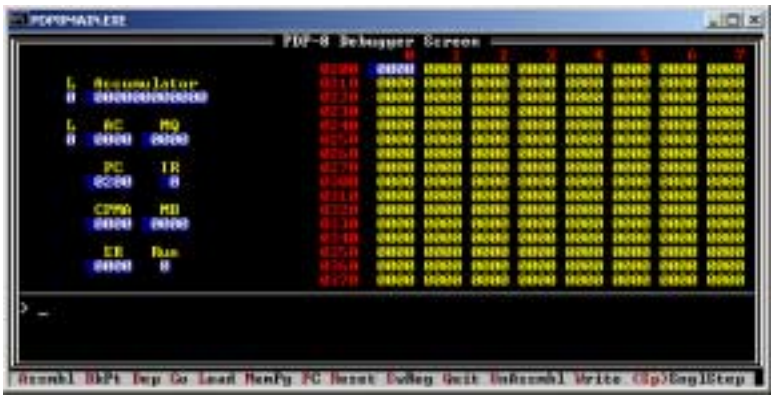


Fig. 2. Debug Screen Display

All CPU Registers appear on the upper left. The first row displays the Link bit (used for a carry out) and the 12-bit Accumulator. On the second row are octal displays for the same Link Accumulator pair plus the Multiplier-Quotient register. Next is the Program Counter and the three-bit Instruction Register followed by the Central Processor Memory Address register and Memory Buffer register followed by the Switch Register and Run bit on the fifth row.

One 128-word page of memory is displayed *in octal* on the upper right: 16 rows of 8 words with row and column headers for addressing. **PgUp** and **PgDn** can be used to display previous and successive pages of memory. The highlighted position is the *memory cursor*, which can be moved using the arrow keys. Typing an octal number will insert it at the *memory cursor* whose address always appears in the Program Counter. Figure 2 displays page 1 of memory: words 0200 (octal) through 0377.

Unlike the other display screens, commands for the Debug screen display are *typed* at the `A>@` prompt in the command area at the bottom of the display. Debug commands are single letters followed by octal values where appropriate (or an assembler instruction for the assemble command). All integers are assumed to be octal.

The bottom line lists thirteen debug commands: **a** followed by an assembler instruction to **Assemble** a line of code at the *memory cursor*, **b** to set a **Break Point** at the current position of the *memory cursor* or to clear the current breakpoint, **d** followed by an octal value to **Deposit** a value to memory (the d is optional), **g** to **Go** (execute a program) starting with the instruction whose address is contained in the Program Counter (*memory cursor*), **l** to **Load** a program into memory

(see Write option below), **m** followed by an octal address to display the **Memory Page** for that address, **p** followed by an octal address to set the **Program Counter** (which also moves the *memory cursor* and displays the corresponding memory page), **r** to **Reset** memory and registers to zero, **s** followed by an octal value to enter a value into the **Switch Register**, **q** to **Quit** the Debug screen ((Esc) also works), **u** to **Unassemble** the value at the *memory cursor* (useful for translating machine instructions to assembler), **w** to **Write** the contents of memory to a file (which can be re-loaded - see Load above), and **(space)** to **single step** (trace) through a program. On-line help using [F1] provides more details about commands.

The Write command dumps the contents of PDP-8 memory to an ASCII-text file. The Load option can be used to load this file back to PDP-8 memory. Using the format mmmm/nnnn where mmmm is the address and nnnn is the contents, the Write command outputs to a file each non-zero word in PDP-8 memory. The final line output is the value of the Program Counter written as $nnnn. Any PDP-8 object code file in this format whether generated by the Write command or created by an ASCII-text editor can be loaded using the Load command.

2.3 The Editor/Assembler

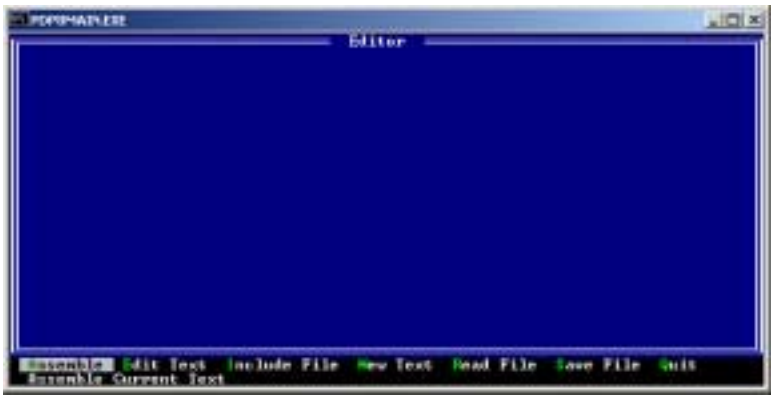


Fig. 3. Editor/Assembler Screen Display

The Editor/Assembler screen display allows the user to create, edit, debug, and assemble PDP-8 Assembler Language (pal) programs. Selecting Edit Text puts the user in Edit mode (the cursor appears in the text area) where the user can enter and edit assembler language programs. The built-in editor is simple and obvious to use but does not have cut and paste capabilities. Use (Esc) or [F10] to exit Edit mode.

Assemble uses a two-pass assembler to assemble the current contents of the text window directly to PDP-8 memory (linkers and loaders are not needed). A successful assemble will allow the user the option of creating a list file which cross lists source code with object code. Successfully assembled code in memory can be executed from the Debug screen display or the Run PDP-8 screen display. A syntax error will halt the assembly process putting the user in edit mode and placing the cursor on the line (but not at the point in the line) where the error occurred. This happens for each syntax error (the process being similar to the way Borland Turbo Pascal compilers found and reported syntax errors).

Edit Text puts the user in edit mode. **Include File** can be used to insert a text file at the cursor position in the text area. **New Text** clears the text area and puts the user in edit mode. **Read File** reads a text file into the text area after first clearing it. **Save Text** will save the contents of the text area. Path names used by the **Include, Read**, and **Save** commands must conform to MS-DOS file naming conventions .

2.4 The Run PDP-8 Program Screen

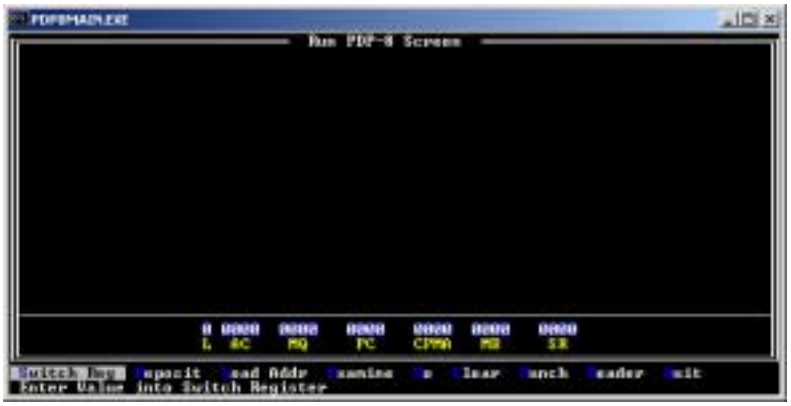


Fig.4. Run PDP-8 Program Screen Display

The Run PDP-8 screen display provides an I/O interface in which to execute PDP-8 programs with the upper portion functioning as a 20 line display. Only an executing PDP-8 program can generate output to this display.

The contents (in octal) of seven registers are displayed. These are some of the register displays that appeared on the PDP-8 console.

Of the nine selections the first five (**SwitchReg** through **Go**) correspond to *switches* found on a PDP-8 console. Their use is discussed below. The sixth, **Clear**, clears the screen display and sets all registers to 0 leaving the contents of PDP-8 memory intact. Since a standard PDP-8 had a paper tape reader/punch attached to it, **Punch** and **Reader** are used simulate paper tape I/O using text files. This gives the PDP-8 Emulator Program a limited file I/O capability.

SwitchReg: The PDP-8 Switch Register was a set of 12 toggle switches on the PDP-8 console that could be used to manually enter numbers directly into the PDP-8. When this option is selected the user is prompted for a four-digit octal integer.

Deposit: This deposits the contents of the Switch Register to the Memory Buffer (MB) register and then performs a memory write to write the contents of the MB register to the memory address given by the Central Processor Memory Address (CPMA) register. The CPMA and Program Counter (PC) registers are both incremented. This is used to deposit values to memory from the PDP-8 console.

AddrLoad: This deposits the contents of the Switch Register to the CPMA register and the Program Counter (PC). This is used to set up the CPMA in preparation for a Deposit or Examine operation.

Examine: This fetches the contents of memory at the address contained in the CPMA register, placing it in the MB register. The CPMA and PC registers are both incremented. This is used to examine the contents of memory from the PDP-8 console.

The SwitchReg, Deposit, AddrLoad and Examine switches were included to make the PDP-8 Emulator Program more realistic. On a physical PDP-8, an operator could use the SwitchReg, AddrLoad and Deposit switches to enter a small program (like a loader) directly into PDP-8 memory. Using SwitchReg, AddrLoad and Examine an operator could also examine (verify) the contents of memory. Both of these actions can be simulated by the PDP-8 Emulator Program.

Another action performed from the PDP-8 console is to set the Program Counter to the starting address of a program. The Switch Register can be loaded with the starting address, the AddrLoad switch can be used to copy the contents of the Switch Register to the Program Counter and the Go option (below) can be used to start execution.

Go: This starts execution of the program in PDP-8 memory beginning with the instruction whose address is contained in the PC register. Execution continues until either a halt instruction is encountered or the user hits **Ctrl/C** (abort).

Clear : This clears the screen and the contents of all registers. Memory is untouched.

3. A QUICK OVERVIEW OF PDP-8 ARCHITECTURE

The following sections provide a quick introduction to the architecture of the PDP-8 and to PDP-8 Assembler Language (PAL) programming. The sections are brief and do not cover all details but knowledgeable readers should be able to learn enough to write and execute simple PDP-8 programs using the PDP-8 Emulator Program.

The memory of the PDP-8 consists of 4096 (2^{12}) twelve-bit words. Bits in a word are numbered *left to right* 0 - 11 with bit 0 being the *most significant* bit.

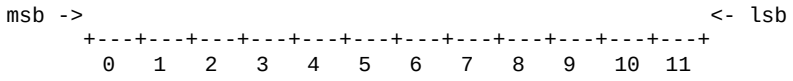


Fig. 5. Bit Numbering for a PDP-8 Word

Memory is partitioned into 32 pages of 128 words each. A 12-bit address is divided into a five-bit page number (bits 0 - 4) and a seven-bit offset (bits 5 - 11). For example address 0200o is page 01o offset 000o (a postfix ‘o’ denotes octal). The reason for this page/offset scheme will be clearly seen when the format of memory reference instructions is examined.

There is a single 12-bit accumulator (AC) with a one-bit link register (L) that captures any carry out of the accumulator. Many PDP-8 instructions make implicit use of the link-accumulator pair.

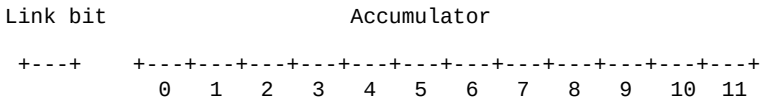


Fig. 6. The Link Accumulator Pair

A 12-bit multiply-quotient register (MQ) is used by multiplication and division operations in PDP-8 models augmented with the Extended Arithmetic Element. EAE instructions are not implemented by the PDP-8 Emulator Program.

Special purpose registers include a 12-bit Switch Register (SR) used to enter values from the console of the PDP-8, a 12-bit Program Counter (PC) which holds the address of the next instruction to be executed, a three-bit Instruction Register (IR), a 12-bit Central Processor Memory Address register (CPMA), and a 12-bit Memory Buffer register (MB). The last two registers are used by the CPU to access memory with the former containing the address and the latter containing the value to be read or written.

The PDP-8 has eight opcodes and three instruction formats. Opcodes 0 – 5 are memory reference instructions (MRI), opcode 6 is a family of I/O instructions, and opcode 7 is a set of orthogonal microinstructions that operate on the Link-Accumulator pair.

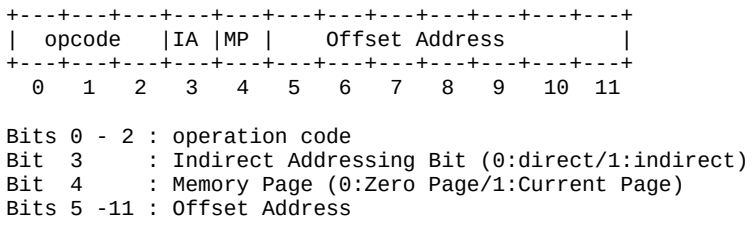
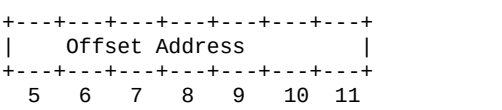

| opcode |IA |MP | Offset Address |
0 1 2 3 4 5 6 7 8 9 10 11
Bits 0 - 2 : operation code
Bit 3 : Indirect Addressing Bit (0:direct/1:indirect)
Bit 4 : Memory Page (0:Zero Page/1:Current Page)
Bits 5 -11 : Offset Address

Fig. 7. Memory Reference Instruction Format

The MRI instruction format only has room for a seven-bit offset address (hence the page/offset addressing scheme). To obtain a 12-bit address either five zeros are prefixed to the offset *(zero page* addressing is indicated by clearing bit 4 to 0) or the five leading bits of the address of the instruction is prefixed to the offset (*current page* addressing is indicated by setting bit 4 to 1). This means that any MRI instruction can directly access 256 out of 4096 words. Indirect addressing is used (bit 3 equals 1) to access any other word in memory.

In the descriptions below, EAddr is short for *Effective Address* and C( ) means *contents of*. Branch instructions are implemented by changing the contents of the PC register.

Opcodes 0 – 5: Memory Reference Instructions



Mnemonic	Opcode	Full Name: Description
AND	0	Bit-wise And: C(AC) = C(AC) \ C(EAddr)
TAD	1	Twos complement Add: C(AC) = C(AC) + C(EAddr); on carry out complement Link
ISZ	2	Increment and Skip on Zero: C(EAddr) = C(EAddr) + 1; if 0 then skip next instruction: C(PC) = C(PC) + 1;
DCA	3	Deposit and Clear Accumulator: C(EAddr) = C(AC); C(AC) = 0
JMS	4	Jump to Subroutine: C(EAddr) = C(PC); C(PC) = EAddr + 1
JMP	5	Jump always: C(PC) = EAddr

Table 1. Six Memory Reference Instructions

Note: A load is done by the TAD instruction after first zeroing out the accumulator using CLA: Clear Accumulator (see opcode 7 instructions below). DCA, Deposit and Clear Accumulator, is a *destructive* store. By itself ISZ (Increment and Skip on Zero) can be used to increment a value in memory or when followed by a JMP (Jump Always) instruction it can be used to implement a counting loop by incrementing a negative count to zero. The JMS Jump to Subroutine instruction saves the return address at the first location in the subroutine and branches to the second location. A return is accomplished by an indirect jump.

Opcode 6 is a family of I/O instructions. The opcode 6 instruction format contains two Aoperand@ fields: a *device number* field in bits 3 - 8 (e.g. device 03 is a keyboard, device 04 is a display screen) and an *extended function* field in bits 9 - 11 which specifies an operation for the device

(e.g. function 001 is skip if device flag set). I/O on the PDP-8 is somewhat complicated and will be covered in detail in a later section.

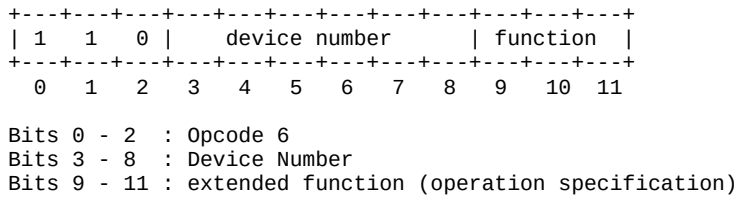
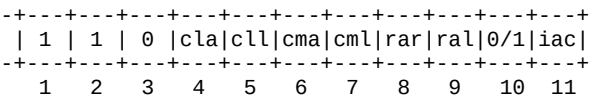

| 1 1 0 | device number | function |
0 1 2 3 4 5 6 7 8 9 10 11
Bits 0 - 2 : Opcode 6
Bits 3 - 8 : Device Number
Bits 9 - 11 : extended function (operation specification)

Fig. 8. Opcode 6 Instruction Format

Opcode 7, like opcode 6, is a family of *micro-instructions* that include instructions to test, increment, complement, and rotate the link accumulator pair. Each bit in the extended opcode field (bits 3 - 11) independently controls a different function. These functions can be combined to create a class of fairly powerful instructions. Opcode 7 instructions fall into three groups. The first group contains functions to clear, complement, increment and/or rotate the link accumulator pair. The execution of functions within a group is prioritized to prevent ambiguities. For example, a clear operation will occur before a complement operation. Group two contains two sub-groups of conditional skip operations that are used for conditional branches. Group three functions use the MQ register and include extended opcodes for the optional Extended Arithmetic Element. The EAE implements in hardware an extended set of operations including multiplication and division. These operations are not implemented by the PDP-8 Emulator Program.

Opcode 7 Group One Microinstructions: bit 3 = 0

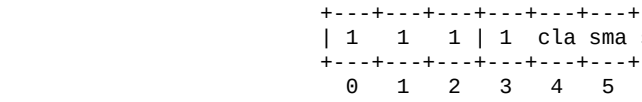


Mnemonic	Opcode	Full Name: Description (priority)
NOP	7000	No Operation
CLA	7200	Clear Accumulator (1)
CLL	7100	Clear Link (1)
CMA	7040	Complement Accumulator (2)
CML	7020	Complement Link (2)
IAC	7001	Increment Accumulator (3)
RAR	7010	Rotate Link Accumulator pair Right 4)
RTR	7012	Rotate Link Accumulator pair Right Twice (4)
RAL	7004	Rotate Link Accumulator pair Left (4)
RTL	7006	Rotate Link Accumulator pair Left Twice (4)

Table 2. Opcode 7 Group One Microinstructions

Note: Priority numbers 1 - 4 determine the order in which combined instructions are executed. Priority 1 instructions (e.g. CLA) are executed before priority 2 (e.g. CMA) etc. The combination CLA CMA would clear the accumulator then complement it thereby setting all bits to 1. Some combinations have their own mnemonics. For example CMA IAC: Complement Accumulator and Increment Accumulator (equivalent to negating the accumulator) is given the mnemonic CIA (7041): Complement and Increment Accumulator. STL: Set Link to One is the mnemonic for CLL CML. GTK: Get Link (put the link in bit 11 of the accumulator) is the mnemonic for CLA RAL.

Opcode 7 Group Two Microinstructions: bit 3 = 1 and bit 11 = 0

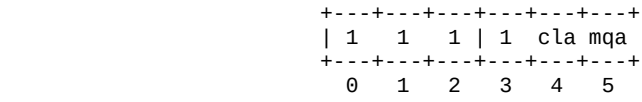


Mnemonic	Opcode	Full Name: Description (priority)
SMA	7500	Skip on Minus Accumulator (1)
SZA	7440	Skip on Zero Accumulator (1)
SNL	7420	Skip on Non-zero Link (1)
SPA	7510	Skip on Positive Accumulator (1)
SNA	7450	Skip on Non-Zero Accumulator (1)
SZL	7430	Skip on Zero Link (1)
SKP	7410	Skip Always (1)
CLA	7600	Clear Accumulator (2)
OSR	7404	Or Switch Register with Accumulator (3): C(AC) = C(SW) ˫ C(AC)
HLT	7402	Halt (4)

Table 3. Opcode 7 Group Two Microinstructions

Note: The first three instructions SMA, SZA and SNL are the conditional skip instructions. The second three instructions SPA, SNA and SZL are obtained by inverting the logic of the first three (indicated by bit 8 = 1). Combinations of SMA, SZA and/or SNL will skip if at least one condition is true. For example SMA SZA will skip if AC is less than or equal to 0. Combinations of SPA, SNA, and/or SZL will skip if all conditions are true. For example SPA SNA will skip if AC is greater than 0. Combinations between the two sub-groups are neither permitted nor necessary.Like Group One, some Group Two combinations have their own mnemonics. LAS: Load Accumulator with Switch Register is the mnemonic for CLA OSR.

Opcode 7 Group Three Microinstructions: bit 3 = 1 and bit 11 = 1



Mnemonic	Opcode	Full Name: Description (priority)
CLA	7601	Clear Accumulator (1)
MQA	7501	Or Accumulator with MQ register (2): C(AC) = C(AC) ˫ C(MQ)
MQL	7420	Load MQ register from Accumulator and Clear Accumulator (2): C(MQ) = C(AC); C(AC) = 0
SWP	7521	Swap AC and MQ registers (3)
CAM	7621	Clear AC and MQ registers (3)

Table 4. Opcode 7 Group Three Microinstructions

Note: Extended Arithmetic Element operations which are not supported by the PDP-8 Emulator Program would appear here as Opcode 7 Group Three microinstructions.

4. PDP-8 MEMORY REFERENCE INSTRUCTION (MRI) ADDRESSING MODES

4.1 Zero/Current Page Direct Addressing:

Because the format of a PDP-8 memory reference instruction only has room for a seven-bit operand field, zero page and current page addressing are used to obtain a 12-bit effective address. The addressing mode is determined by bit 4 of the instruction. With zero page addressing (bit 4 = 0) five zeros are concatenated to the front of the seven-bit offset to obtain a 12-bit address. With current page addressing (bit 4 = 1) the leading five bits (page number) of the address of the MRI instruction is concatenated to the seven-bit offset. Thus an MRI instruction can only directly reference an operand that is either on page zero or on the same page as the instruction itself. This means that most operands have to be fairly close to the MRI instruction that references them.

Example (zero page addressing): Address 6424o contains the instruction 1167o (written 6424o/1167o). Since bit 4 of the instruction is zero, the effective address of this TAD instruction is obtained by concatenating five zeros to the seven-bit offset for an effective address of 0167o.

```
 1   1   6   7          op  I M    offset
001 001 110 111   ->   001 |0|0| 1 110 111
                                    0   1   6   7
effective address -> 00000 + 1 110 111 = 000 001 110 111
```

Fig. 9. Zero Page Addressing Example

Example (current page addressing) : Given 6424o/1367o where bit 4 of the instruction is 1, the effective address of this TAD instruction is obtained by concatenating the leading five bits of the address of the instruction (110 10 from 110 100 010 100 = 6424o) to the seven-bit offset for an effective address of 6567o.

```
 6   4   2   4                          1   1   6   7
110 100 010 100 <- address/contents -> 001 001 110 111
110 01 <- page number                    1 110 111 <- offset

effective address -> 110 10  1 110 111 = 6567o
```

Fig. 10. Current Page Addressing Example

Since page zero is accessible from any address in memory, global variables are stored on page zero. Local variables should be stored on the current page.

4.2 Indirect Addressing

Since zero/current page direct addressing can only access two pages out of 32, the PDP-8 uses indirect addressing to access the other 30 pages (bit three equals 1). Here the zero/current page address calculation yields *not the effective address* but the *address of the effective address*.

4.3 Auto Index Addressing

Memory locations 0010o - 0017o (on page zero) function as auto-index registers. Whenever addresses 0010o - 0017o are accessed *using indirection* their contents are FIRST incremented then used as the effective address.

Example (auto-index addressing): Let address 0010o contain the value 3407o. The instruction 1410o references auto-index register 0010o indirectly (bit 3 = 1, bit 4 = 0, offset = 010o). Therefore we increment the contents of index register 0010o and apply indirection; the effective address is 3410o.

Examples using each of these addressing modes are given in the following sections.

5. PDP-8 ASSEMBLER LANGUAGE (PAL) PROGRAMMING

5.1 The Basics

The format of a PDP-8 Assembler Language instruction has up to five fields

```
symbolic address, opcode(s) i offset /comment
```

Commas terminate a symbolic address (label), an "i" denotes indirect addressing and a "/" indicates that a comment follows. All fields are optional except for opcodes (note the plural since opcode 7 microinstructions can be combined). For example, the following PAL code performs the operation c = a - b.

```
/ code section
*200           / code begins at 0200o
Main, cla cll  / clear AC and Link
      tad B    / load B
      cia      / negate it
      tad A    / add A : AC = A - B
      dca C    / store at C
      hlt      / halt
      jmp Main / to continue go to Main
/ data section
*0300         / data begins at 0300o
A, 17         / 17 octal = 15 decimal
B, 22         / 22 octal = 18 decimal
C, 0
$Main
```

Fig. 11. A PAL program to compute c = a + b

Note: To load the accumulator, first clear it (CLA CLL) then add in (TAD) the value to be loaded. The CIA (Complement and Increment Accumulator) instruction negates the accumulator. Recall that CIA is a mnemonic for CMA IAC.

An *offset* can be a number, a symbolic address, a dot "." (used to denote the current address of the instruction) or a simple expression using the above. For example `jmp .-6` could be used in place of `jmp Main` and since the address of B is one more than the address of A `tad A+1` could be used in place of `tad B`.

To allocate storage for variables and constants use a symbolic address, (comma) followed by the value (use 0 if you don=t care about the initial value). By default all values are octal. To specify a decimal use a suffixed "d" or use an 8 or 9 as a digit (see example following). Data should be located on the same page as code or on page zero for global variables.

```
A,   15d       / decimal 15
B,   18        / decimal 18 since 8 is not an octal digit
C,   0
```

The directive **nnnn* where *nnnn* is any octal value sets the value of the *location counter* used by the assembler. It is used to specify where assembled code is placed in memory (the PDP-8 Emulator Program generates non-relocatable code). The directive $*label* indicates the end of the program and specifies *label* as the entry point (address of the first instruction) for the program.

Symbolic addresses (identifiers and labels) must begin with a letter and contain only letters and digits and should not exceed eight characters. Special characters and embedded blanks are not permitted. The letter `Ai@` (or `AI@`) is *reserved* for indirect addressing. Symbolic addresses are not case sensitive.

The default radix is 8 and all numbers are considered octal unless they contain an 8 or 9 or have a suffixed "d" (decimal). A suffixed "o" (octal) is also permitted but not necessary. The directives **decimal** and **octal** can be used to change the default radix beginning at the line containing the directive.

A number of characters have special meaning or use

Symbol	Use/Meaning
I, i	reserved for indirect addressing
/ (slash)	denotes that a comment follows
, (comma)	terminates a symbolic address but is not part of the symbolic address
. (dot)	denotes the current address of the instruction
* (asterisk)	sets the value of the location counter
$ (dollar sign)	indicates the end of the program
; (semi-colon)	terminates a statement; used for multi-statement lines
= (equal sign)	equates a symbol to a value (e.g. `x=-1` sets the symbol x equal to -1) This does not allocate storage.
› (single quote)	used to indicate a character (e.g ›A-) or a string (›Hello-). They should occur in pairs

Table 5. Reserved Words and Symbols

In the code below a counting loop is used to sum the integers from 1 to N. It first initializes X to 0 then uses a loop to increment and add it to Sum. The counting loop is controlled by the variable Count which is initialized to minus N then incremented on each pass through the loop until it equals 0. ISZ instructions are used for loop control and to increment X. By convention most PAL programs begin on page one at location 0200o.

```
/
/ Code Section
/
*0200               / Code starts at address 0200
Main, cla cll       / clear AC and Link
      tad N         / load N
      cia           / negate it
      dca Count     / deposit it to Count and clear AC
      dca X         / clear X (AC is zero)
      dca Sum       / clear Sum (AC is zero)
Loop, isz X         / add 1 to X
      tad Sum       / load Sum (assumes AC is 0)
      tad X         / add in X
      dca Sum       / store result in Sum (AC cleared)
      isz Count     / increment Count and skip on zero
      jmp Loop      / otherwise loop
      hlt           / done
      jmp Main      / allows easy restart
/
/ Data Section
/
*0300               / Data starts at address 0300
N,     12           / N equals 10 decimal
Count, 0            / Loop Counter
X,     0            / Added to Sum
Sum,   0            / Running Sum
$Main               / end of pgm - entry point
```

Fig. 12. A PAL program to sum the integers from 1 to N

To execute this program do the following:

1. Start the PDP-8 Emulator Program and go to the Editor/Assembler screen. Go into Edit mode (Edit Text) and enter the above code as is. When done, use (Esc) or [F10] to exit Edit mode.
2. Select Assemble to assemble the code to memory. If there are errors, fix them and try again until you get a clean compile. Creating a list file is optional. You may want to save your source code to a file.
3. Quit the Editor/Assembler and go to the Debug screen display. Since this PDP-8 program does no I/O, you can only see the results of the execution from the Debug screen. When you enter the Debug screen display you should see the *memory cursor* positioned at address 0200o of page 1 of memory. If not, use the arrow keys to move it so the PC contains the correct starting address.
4. To execute the code, at the > prompt in the Command window type `g(Enter)`. If the program executes correctly you should see the result, 67o (55d), at address 0303o. The *memory cursor* should be highlighting the 5200o instruction (jmp Main) at address 0215o.
5. To re-execute the program either use the arrow keys to reposition the *memory cursor* at address 0200o or start where the previous execution left off at address 0215o with the `jmp Main` instruction. If you *single step* through the program using the (space) bar, observe how the accumulator changes. You may want to reset Sum (address 0303o) to zero and/or enter a different value for N (address 0300o) before re-executing.

Since the purpose of this program is to demonstrate various PDP-8 instructions, it is not very efficient in summing the integers from 1 to N!

5.2 More Examples of PDP-8 Assembler Programs

Since a good way to learn PDP-8 assembler is to study examples of PDP-8 programs, three additional programs are presented that demonstrate a variety of programming and addressing techniques. The first program separately sums the odd and even integers between 1 and 100o (64). It's similar to the previous program except that it tests if X is odd or even and uses separate variables (Odd, Even) to hold each sum. After it loads an integer X into the accumulator the RAR (Rotate Accumulator and Link Right) instruction rotates the least significant bit into the Link which is then tested by the SNL (Skip on Non-Zero Link) skip instruction. If the Link bit is 1 (an odd integer) the JMP (jump) instruction following is skipped and the following RAL (Rotate

Accumulator and Link Left) instruction restores X which is then added to the variable Odd. Otherwise the JMP instruction is not skipped and control branches to a different code segment that also uses the RAL instruction to restore X but adds it to Even. At the end the value of Odd (address 0303o) should be the sum of the odd integers between 1 and 63 (which is 32^2 = 1024 or 2000o) and the value of Even (address 0304o) should be the sum of the even integers between 2 and 64 (which is 32(32+1) = 1056 or 2040o). This code demonstrates PDP-8 conditional branching which is done by pairing a "test and skip" instruction with a jump instruction. It also uses "dot" notation to assemble the target addresses for jump instructions.

```
/
/ Code Section
/
*0200              / code starts at address 0200
main,   cla cll    / clear AC and Link
        tad N      / load N
        cia        / negate it
        dca Count  / deposit it to Count and clear AC
        dca X      / clear X (AC is zero)
        dca Even   / clear Even (AC is zero)
        dca Odd    / clear Odd (AC is zero)
Loop,   isz X      / add 1 to X
        cla cll    / clear AC and Link
        tad X      / load X
        rar        / rotate least significant bit into Link
        snl        / skip on non-zero link (X is odd)
        jmp .+5    /    else X is even
        ral        / restore X
        tad Odd    /    and add to
        dca Odd    /    Odd
        jmp .+4    / jump to end of loop
        ral        / restore X
        tad Even   /    and add to
        dca Even   /    Even
        isz Count  / increment Count and skip on zero
        jmp Loop   / otherwise Loop
        hlt        / done
        jmp Main   / allow easy restart
/
/ Data Section
/
*0300              / Data starts at address 0300
N, 100             / N equals 64
Count, 0           / Loop Counter
X, 0               / added to Odd or Even
Odd, 0             / Sum of Odds
Even, 0            / Sum of Evens
$Main
```

Fig. 13. A PAL program to separately sum even and odd integers

Note: The `cla cll` instruction at `Loop+1` is not necessary since the AC will always be cleared at this point.

The next program separately sums the negative and positive values in an array. Indirect addressing is required to access the elements of the array, which for this program is kept on a different page.

Indirect addressing requires that the address of the array be stored in a variable Addr then loaded into a second variable Ptr which is incremented (using ISZ) at the end of each pass through a loop. As each element in the array is accessed it is tested for being greater than zero using the SPA SNA (Skip on Positive Accumulator and Skip on Non-Zero Accumulator) combination. If true the jump instruction following the test is skipped and the value is added to PosSum; otherwise the jump statement transfers control to a segment of code, which adds the value to NegSum. The program loops until a zero entry in the array is found.

```
/
/ Code Section
/
*0200              / code starts at address 0200
main,   cla cll    / clear AC and Link
        dca Count  / clear Count  (AC is zero)
        dca PosSum / clear PosSum (AC is zero)
        dca NegSum / clear NegSum (AC is zero)
        tad Addr   / get address of array
        dca Ptr    /    and store in Ptr
Loop,   cla cll    / clear AC and Link
        tad i Ptr  / get A[Ptr]
        sna        / if A[Ptr] != 0 then continue
        jmp Done   /    else done
        spa sna    / test if A[Ptr] > 0
        jmp .+4    /    else A[Ptr] < 0
        tad PosSum / add to PosSum
        dca PosSum / store PosSum
        jmp .+3    / jump to increment Ptr
        tad NegSum / add to NegSum
        dca NegSum / store NegSum
        isz Ptr    / increment Ptr
        isz Count  / increment Count
        jmp Loop   / go again
Done,   hlt        / done
        jmp Main   / allow easy restart
/
/ Data Section
/
*0300              / Data starts at address 0300
Addr, A            / address of array A
Ptr, 0             / pointer to item in array
Count, 0           / Count of items in array
NegSum, 0          / Sum of negatives
PosSum, 0          / Sum of positives
/
/ Array
/
*0400              / Array A starts at address 0400 - Page 2
A,  -2; 1; 5; 3; -3; -1; -5; 2; 4; -4; 0
$Main
```

Fig. 14. A PAL program to separately sum the positive and negative integers in an array

The third program, a variant of the previous, uses auto-index addressing (with auto-index register 0010o) to access elements in the array. Since auto-index addressing first *increments* the contents of the auto-index register *before* applying indirection, auto-index register 0010o must be loaded

with the address of the array minus 1. This value, the address of the array minus 1, is stored on page zero making the array accessible from any page.

```
/
/ Code Section
/
*0200              / code starts at address 0200
main,   cla cll    / clear AC and Link
        dca Count  / clear Count  (AC is zero)
        dca PosSum / clear PosSum (AC is zero)
        dca NegSum / clear NegSum (AC is zero)
        tad Addr   / get address of array - 1
        dca 10     /    and store in auto-index register 10
Loop,   cla cll    / clear AC and Link
        tad i 10   / get A[i]
        sna        / if A[i] = 0
        jmp Done   /    then done
        spa        / test if A[i] > 0
        jmp .+4    /    else A[i] < 0
        tad PosSum / add to PosSum
        dca PosSum / store PosSum
        jmp .+3    / jump to increment Count
        tad NegSum / add to NegSum
        dca NegSum / store NegSum
        isz Count  / increment Count
        jmp Loop   / go again
Done,   hlt        / done
        jmp Main   / allow easy restart
/
/ Data Section
/
*0100              / global variables stored on page 0
Addr, A-1          / address of array A - 1
*0300              / local variables stored starting at address 0300
Count, 0           / Count of items in array
NegSum, 0          / Sum of negatives
PosSum, 0          / Sum of positives
/
/ Array
/
*0400              / Array starts at address 0400 - Page 2
A, -2; 1; 5; 3; -3; -1; -5; 2; 4; -4; 0
$Main
```

Fig. 15. A PAL program using auto-index addressing to access array elements

Since none of these programs do I/O, they must be run in the Debug screen display to see results. PDP-8 I/O will be discussed below following the section on PDP-8 subroutines.

6. CALLING PDP-8 SUBROUTINES

Since we have not discussed how to do I/O on the PDP-8, all of the sample programs presented thus far were run from the Debug screen display to see results. Simple PDP-8 I/O will be discussed in the section following this one on subroutines since we will consolidate I/O code into subroutines.

The JMS (Jump to Subroutine) instruction works by first storing the return address at the first address of the subroutine and then branching control to the second location. A return is made by an indirect jump using the return address stored at the first location. In the sample program below, the subroutine Abs which returns the absolute value of the accumulator is used to compute c = |a-b|. Abs is called by the jms instruction at Main+4.

```
/
/   Code Segment
/
*0200            / code starts at address 0200
Main, cla cll    / clear AC and link
      tad B      / Load B
      cia        / Negate B
      tad A      / A - B
      jms Abs    / take absolute value
      dca c      / store results at C
      hlt        / halt
      jmp Main   / to continue - goto entry point
/
/   Subroutine
/
*0250              / subroutine starts at address 0250
Abs, 0             / store return address here
     sma           / skip on minus accumulator
     jmp .+2       / otherwise jump current address plus 2
     cia           / negate accumulator
     jmp i Abs     / return via indirect jump
/
/   Data Segment
/
*0300           / data starts at address 0300
A, 17
B, 22
C, 0
$Main           / entry point is Main
```

Fig. 16. A PAL program to compute c = |a – b|

The Abs subroutine is on the same page as the subroutine call; it cannot be called from a different page *unless indirection* is used. To execute an *indirect* subroutine call, store the *address* of the subroutine at a location AdrAbs on page zero and use the *indirect* subroutine call `jms i AdrAbs`. Examples of indirect subroutine calls are given in the next section.

7. DOING SIMPLE PDP-8 I/O

7.1 Programmed I/O with busy waiting

Programmed I/O using opcode 6 instructions is the simplest method of doing I/O on the PDP-8. The format of an opcode 6 instruction has a six-bit device number field and a three-bit function field. The PDP-8 Emulator Program only supports two I/O devices: a keyboard (device number 03) for input and a printer, actually a display screen, (device number 04) for output. The list of

opcode 6 I/O instructions implemented on the PDP-8 Emulator Program for these two devices is given below

Opcode 6 Instruction Format

```
+---+---+---+---+---+---+---+---+---+---+---+---+
| 1   1   0 |    device number      | function  |
+---+---+---+---+---+---+---+---+---+---+---+---+
  0   1   2   3   4   5   6   7   8   9   10  11
```

Mnemonic	Opcode	Full Name: Description
KFC	6030	Clear Keyboard Flag:
KSF	6031	Skip on Keyboard Flag Set
KCC	6032	Clear Keyboard Flag and Accumulator
KRS	6034	Read Keyboard buffer Static: AC4..AC11 = AC4..AC11 or Keyboard Buffer
KRB	6036	Read Keyboard Buffer dynamic: C(AC) = 0; Keyboard Flag = 0; AC4..AC11 = AC4..AC11 = AC4..AC11 or Keyboard Buffer
TFL	6040	Set Printer Flag
TSF	6041	Skip on Printer Flag Set
TCF	6042	Clear Printer Flag
TPC	6044	Load Printer Buffer with Accumulator and Print: Printer Buffer = AC4..AC11
TLS	6046	Load Printer Sequence: Printer Flag = 0; Printer Buffer = AC4..AC11

Table 6. Opcode 6 I/O Instructions

I/O on the PDP-8 is done by transferring eight-bit ASCII characters between the right-most eight bits of the Accumulator (AC4..AC11) and an eight-bit I/O device buffer (keyboard or printer display).

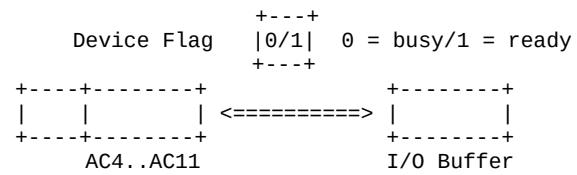


Fig. 17. Accumulator – Device Flag – Device Buffer Schematic

A device flag bit handles synchronization between the CPU and I/O device. When the I/O device is *busy* the flag is cleared to 0; when the device is *ready*, the flag is set to 1.

To perform I/O the CPU executes a *wait loop* that checks the device flag. When the I/O device is ready to send/receive a character it sets the device flag to 1. The CPU detects that the device flag is set using a Askip on flag set@ instruction and breaks out of the wait loop. Once out of the wait loop the CPU transfers the byte in the 8-bit device buffer to/from the right most 8 bits of the accumulator and clears the device flag using a Atransfer and clear flag@ instruction.

While ten opcode 6 I/O instructions are supported by the PDP-8 Emulator Program, only five seem to be used: two for output to the printer display: TSF - Skip on Printer Flag Set and TLS - Load Printer Sequence and three for input from the keyboard: KSF - Skip on Keyboard Set, KCC - Clear Keyboard Flag and KRB - Read Keyboard Buffer. I/O is best done with code consolidated into subroutines. The following program demonstrates the standard way to read and write characters by reading and echoing a character typed from the keyboard. It uses two I/O subroutines: GetChar and Type. GetChar reads a character from the keyboard returning it in the accumulator. Type transfers an eight-bit character in the accumulator to the output buffer for display. Both subroutines begin by entering a wait loop that repeatedly checks if the device flag is set (ready). When the device flag is set control breaks out of the loop allowing both to execute a transfer instruction to move the byte between the device buffer and the accumulator. The program terminates when the user hits a carriage return (ASCII code 13) .

```
/
/ Main Code Segment
/
*0200
Main, cla cll        / clear AC and link
      kcc            / clear keyboard flag
      tls            / wake-up printer – display unprintable null
Again,jms i GetChar  / read a character
      dca Ch         / store it
      tad Ch         / get the character and
      tad MCR        /    check if it-s a CR
      sna            / if not skip next instruction
      jmp Done       /    else done
      cla cll        / clear AC
      tad Ch         /    and load character
      jms i Type     / echo to screen
      jmp Again      / go again
Done, hlt
      jmp Main
/
/ I/O routines
/
*0250
GetChar, 0               / return address here
         ksf             / is keyboard flag raised?
         jmp .-1         / no  - loop
         krb             / yes - read character to AC
         jmp i GetChar   / return
Type,    0               / return address here
         tsk             / is printer flag raised?
         jmp .-1         / no  - loop
         tls             / yes - print character
         cla cll         / clear AC and link
         jmp i Type      / return
/
/ Data Segment
/
*0300
Ch,  0             / character
MCR, 13d           / minus ASCII code for CR
$Main
```

Fig. 18. A PAL program to read and echo characters

Use of the KCC instruction at address 0201o to initially clear the keyboard flag seems to be a standard trick. The same seems to be true for the TLS instruction at address 0202o that causes the output device to display an unprintable null character. Storing the negative value of an ASCII code to be tested for (in this case a minus 13 decimal) and adding the value to check for equality is also standard. Finally, GetChar only reads a character; to echo it to the printer display, the program must explicitly call the Type subroutine.

Since this program does PDP-8 I/O it can be executed from the Run PDP-8 Pgm screen display.

7.2 A PDP-8 Hello World Program

The following program displays the string “Hello World”. It also illustrates a number of PDP-8 I/O techniques. First the subroutine Type is located on page 30 and its starting address is stored on page zero. The indirect subroutine call `jms I Type` is used to invoke it. This allows the routine to be called from any page in memory. Next, the string ›Hello World‹ is implemented as a null terminated array of characters. The data structure for the string stores its own address (using the “.” character) followed by the string to be displayed (including the carriage-return line feed character combination 13d 10d) ending with a null character. Storing the address of the string as its first word fits the *increment then indirect* effective address calculation of auto-index addressing. Auto-index addressing using auto-index register 0010o is used to load each character into the accumulator and Type is called to display the character. All of this is embedded in a loop that continues until a null character is detected. Finally to make the code more readable we assign the value 0010o to the symbol IR10 (Index Register 10o) using IR10=10.

```
/
/  Subroutine Vectors
/
*0050     / subroutine entry point stored at address 0050o
Type,     XType
/
/  Code Segment
/
*0200              / Code Segment starts at 0200o
Start, cla cll     / Clear AC and Link
       tls         / Wake Up Printer
       tad STR     / Get address of output string
       dca IR10    /   and store it in Index Register 10
Loop, tad i IR10   / Get next character
      sna          / Skip if not a Null character
      jmp End      /    else end
      jms i Type   / Type it
      jmp Loop     / Loop!
End,  hlt          / Done!
      jmp Start
/
/  Data Segment
/
*0250              / Place String at 0250o
STR, .             / STR stores its own address - string
                   /    data follows
'Hello World';13d;10d;0
IR10=10            / Index Register10 is address 10
/
/ Subroutine Segment - Page 30
/
*7400
XType,    0              / Store Return Address Here
          tsf            / Is Printer Ready?
          jmp .-1        / No - Loop! (Wait Loop)
          tls            / Yes - Print the character!
          cla cll        / Clear AC and Link
          jmp i XType    / Return
$Start
```

Fig. 19. A PDP-8 Hello World Program

8. A PDP-8 BIOS (BASIC INPUT OUTPUT SUBROUTINES)

Given below is a set of four basic I/O subroutines for the PDP-8. This code fits on a single page (page 30) and is made accessible to any page in memory by storing the starting addresses for the subroutines on page zero. The subroutines are called using indirect subroutines calls (e.g. `jms i Type`, not `jms Type`). Recall that direct addressing subroutine calls forces subroutine code to be placed on the zero or current page. Placing subroutine code on every page where it is called causes unnecessary duplication and space limitations restrict the use of page zero for subroutine code.

The sample hello world program below uses three of the four I/O subroutines: Type, PrtStr (which prints a null terminated string of characters) and CRLF (which prints a Carriage Return Line Feed

combination - a newline character). While not elaborate, all four subroutines allow the writing of PDP-8 programs that do limited I/O.

```
/
/ Basic I/O Routine Vectors - Page 0
/
*0050
GetChar,  XGetChar
Type,     XType
CRLF,     XCRLF
PtrStr,   XPrtStr
/
/ Code Segment - Page 1
/
*0200
Main,  cla cll       / clear AC and Link
       kcc           / clear keyboard flag
       tls           / wake up printer
       cla cll    /
       tad Str
       jms i PrtStr
       jms i CRLF
       hlt
       jmp Main
/
/ Data Segment
/
*0300
Str, .                    / Str stores its own address
     ›Hello World!÷ ; 0    / Null Terminates String
/
/ Basic I/O Subroutines - Page 30
/
*7400
XGetChar, 0             / store return address here
        ksf             / Is keyboard flag raised?
        jmp .-1         / no - loop
        krb             / yes - read character to AC
        jmp i XGetChar  / return
XType,  0               / store return address here
        tsf             / is printer ready?
        jmp .-1         / no -loop
        tls             / yes - print character
        cla cll         / clear AC and Link
        jmp i XType     / return
XCRLF,  0               / store return address here
        cla cll
        tad .+5         / get CR
        jms XType       /   and type it
        tad .+4         / get LF
        jms XType       /   and type it
        jmp i XCRLF     / return
        13d;10;         / carriage return; line feed
XPrtStr,0
        dca 10          / store address at index reg. 10
        tad i 10        / get character
        sna             / is it null?
        jmp i XPrtStr   / yes - return
        jms XType       / no - type it
        jmp .-4         / get next character
$Main
```

Fig. 20. A PDP-8 Hello World Program using Basic I/O Subroutines

GetChar reads a character from the keyboard returning it in the accumulator. Type displays the ASCII character stored in the eight right-most bits of the accumulator and clears the accumulator. CRLF outputs a carriage return line feed combination - a newline character. PrtStr (print string) displays a null terminated string whose address is passed to the subroutine using the accumulator. The first word of the string must contain its own address which is why the code sequence `cla cll; tad Str` precedes the `jms i PrtStr` instruction. PrtStr immediately deposits the address to auto-index register 10 then uses auto-index addressing to load each character and checks that it's not null before calling the Type command to display it. Note that the accumulator is used to pass parameters to three of the subroutines.

Code to read and write integers is left to the reader. Reading and writing unsigned 4-digit octal values is fairly easy to do but code to read and write decimal integers requires multiplication and division by decimal 10.

9. PDP-8 MACHINE CODE

Given the simplicity of the PDP-8 architecture, machine code programming is possible although tedious since operand address calculation must be done by hand. Numeric code can be entered directly into the PDP-8 memory from the Debug screen display by positioning the *memory cursor* at the address where you want to enter the code, typing the octal value, and hitting enter. The value will be deposited to that location and the *memory cursor* will advance to the next. For example, the following machine code program computes c = |a – b|. We display the code using the address/contents convention "nnnn/mmmm".

```
0200/7300      clear accumulator and link
0201/1300      load B (from address 0300)
0202/7041      negate (complement and add 1)
0203/1301      add A (from address 0301
0204/7500      skip if negative
0205/5207      jump to address 0207
0206/7041      negate A - B to make positive
0207/3302      deposit |A - B| to C (address 0302)
0210/7402      halt
0300/0017      A
0301/0022      B
0302/0000      C
```

Fig. 21. PDP-8 machine code to compute c = |a-b|

To execute the above machine code, do the following.

1. From the Debug screen display, use the PC command (p 0200) to position the *memory cursor* at address 0200o and display page 1 of memory.
2. Beginning at address 0200o, enter the code as given above (7300 1300 7041 etc). After entering the last instruction 7402 at address 0210o, position the *memory cursor* at address 0300o and enter the data (0017 0022).
3. Reposition the *memory cursor* at address 0200o, the starting address for the code. At the Command Window prompt A>@ hit **g** (enter) to execute the program or **(space**) to single step through the code. The answer will appear in location 0302o

10. ADDITIONAL FEATURES

In the interest of completeness, the following section reviews a number of PDP-8 Emulator Program features in greater detail and introduces some additional features that give it a greater capacity for simulation. There is a more complete description of the Debug screen commands, the capabilities of the Editor and the various options supported under the Run PDP-8 Program screen. Using the Run PDP-8 Program screen, an example is given that simulates using the PDP-8 console to load code directly into memory. The PDP-8 Emulator Program also features a simulated paper tape reader/punch which gives it a limited file handling capability. Programs punched on paper tape could be loaded into PDP-8 memory using a loader program. One such loader program is presented and it is shown how this program can be used to load other programs from paper tape. Finally the PDP-8 could handle interrupts so there is a short discussion on how interrupts are implemented on the PDP-8 Emulator Program along with an example program that makes use of them.

10.1 The Debug Screen

The **Assemble** command can be used to assemble single lines of PAL code at the *memory cursor*. Small programs can be entered and assembled as shown by the following example which computes z = x + y.

Position the memory cursor at address 0300o and at the > prompt enter the commands

```
>a x, 4
>a y, 5
>a z, 0
```

to define three variables x, y, and z. Then reposition the memory cursor at address 0200o and at the > prompt enter the commands

```
>a main, cla cll
>a tad x
>a tad y
>a dca z
>a hlt
>a jmp main
```

The **Unassemble** command to un-assemble machine code to assembler is useful for debugging code. Position the *memory cursor* at the instruction you want unassembled and at the > prompt hit **u** (enter). Unassemble is incorporated into the **Single Step** (space) command which unassembles both the instruction just executed and the instruction to be executed next.

The Break point command (**BkPt**) sets a single program breakpoint at the *memory cursor* or it clears the current breakpoint wherever it is. Consequently to set a different breakpoint, move the *memory cursor* to where you want to set the new one and at the > prompt enter **b** to clear the current break point then enter **b** again to set it (two commands). A breakpoint may also be set using the **Go** command if an octal address is included after the **g** at the > prompt (e.g. >g 210).

The **Deposit** command is used to enter octal values at the *memory cursor*. At the > prompt enter any octal value preceded by a "d" (the "d" is optional!) followed by (enter). The value will be deposited at the *memory cursor*, which automatically advances to the next location.

Write and **Load** commands to save and restore the current contents of the PDP-8 memory require path names that conform to MS-DOS file naming conventions. That means all file and directory (folder) names are restricted to letters, digits and/or underscore characters and are 8 characters or less. For example at the > prompt `>w c:\pgm01.obj` will save the current contents of memory to a text file (I use .obj for object code). The same code may be loaded using `>l c:\pgm01.obj`.

The **Write** command scans PDP-8 memory and saves every non-zero location using the address/contents format nnnn/mmmm. For example, the program assembled using the a command above when saved to a text file using the w command would appear as

```
0200/7300
0201/1300
0202/1301
0203/3302
0204/7402
0205/5200
0300/0004
0301/0005
0302/0011
$0205
```

Fig. 22. Text file from Debug screen w command

The current value of the *memory cursor* is saved as $nnnn. The **Load** command expects a file in the format created by the **Write** command. **Load** does no syntax checking and since it ignores all characters past the ninth, comments may be attached to each line.

PDP-8 programs that do I/O using opcode 6 instructions can be run from the Debug screen display. Output generated by a PDP-8 program will appear in yellow (to distinguish it from user command input) in the Command window. However, when using the space bar to single stepping through a program that is trying to read input, the interface has difficulty in distinguishing between input meant for the PDPB8 program and user command input. To get around this problem, any string of input preceded by a *single quote* is ignored by the command interface and used as input to the PDP-8 program.

Ctrl/C will halt any executing program. Illegal Debug screen commands are ignored; there is no error message.

10.2 The Editor

The text editor is simple and obvious to use but it does not have block, copy, and paste capabilities and lines are restricted to 77 characters in length. Inserting characters into the middle of a line of 77 characters will delete characters at the end; adding characters at the end will start a new line. Two lines can be joined by positioning the cursor on the first character of the second line and hitting backspace. The **Ins** key will switch between insert mode (a flat cursor) and overwrite mode (a block cursor). **Ctrl/Y** will delete an entire line. The **Read, Include**, and **Write** commands prompt the user for file and path names that must conform to MS-DOS file naming conventions. Assembler programs *can be created and edited* using any ASCII text editor (e.g. Notepad) then **Read** or **Included** into the Editor text area to be assembled. **Assemble** will only assemble the current contents of the Edit text area. The assembler does not recognize tab characters that may be

used by other ASCII editors (a tab character appears as an “o” in the Edit text window) so tabs should be avoided. The built in editor does not generate tabs; it inserts spaces for tabs.

10.3 The Run PDP-8 Pgm Screen – Paper Tape Option

A teletype was the standard I/O device for the PDP-8. A teletype could have an 8-hole paper tape reader/punch attached to it. Eight hole paper tape had 8 tracks spanning the width of the tape where the presence ! or absence " of a hole encoded a 1 or 0. Octal values between 000 and 377 could be written to and read from paper tape, which gave the PDP-8 a limited file handling capability - files being paper tape.

```
track          87654321
               --------
               !!!!!!!!      377     <- octal values
               !!!!!!!!      377
               "!""""!"      102
               """""""       000
               ""!!!"!!      073
               """""""       000
               "!""""!"      102
               """""""!      001
               """"!"!"      012
               ""!"!"""      050
```

Fig. 23. Eight -Track Paper Tape

A paper tape reader/punch is simulated on the PDP-8 Emulator Program by the **Punch** and **Reader** options on the Run PDP-8 Screen display. These options allow a PDP-8 program to read from or write to a text file (instead of paper tape). When either or both options are selected the user is prompted for the path name of the text file (MS-DOS file naming conventions required) and for whether the file is ASCII characters or Abinary@. Binary format encodes rows of 8 bits as lines of three digit octal integers between 000 and 377. Binary format was included so that the PDP-8 Emulator Program could simulate using a loader program to load programs from paper tape (discussed below).

When either option is selected, it is highlighted in yellow. The **Reader** option disables PDP-8 program input from the keyboard and the **Punch** option disables PDP-8 program output to the screen.

10.4 The Run PDP-8 Pgm Screen - Program Loading

The PDP-8 Emulator Program allows users to enter code directly into memory from the Debug screen display or to assemble programs to memory from the Editor/Assembler display. Machine

code can also be loaded into memory from the Run PDP-8 Program screen with the **Switch Register**, **Load Addr Deposit**, and **Examine** switches using a technique similar to one used to enter a program from the console of a real PDP-8. To load code by hand into PDP-8 memory do the following

1. Use **Switch Reg** to enter the address of the first instruction into the Switch Register.
2. Use **Load Addr** to copy the contents of the Switch Register to the CPMA register and the Program Counter (PC register).
3. Use **Switch Reg** to enter the first instruction into the Switch Register
4. Use **Deposit** to deposit the contents of the Switch Register to the MB register and execute a memory write which will copy the contents of the MB register to the address given in the CPMA register. The CPMA and PC registers are subsequently incremented.
5. Repeat steps 3 and 4 for all instructions.

To verify the code use **Examine** in place of **Deposit**. This will execute a memory read whereby the contents of memory whose address is in the CPMA register will appear in the MB register where it can be checked. The CPMA and PC registers are subsequently incremented.

10.5 The Run PDP-8 Pgm Screen - The RIM Loader

The following sixteen-word program called the RIM (Read Input Mode) loader was used to load PDP-8 programs from paper tape (see *Introduction to Programming - PDP-8 Handbook Series* p 5-5). It is one of a number of such loaders used by the PDP-8. The RIM loader is first loaded into memory (by hand since it was small?) then paper tapes of other programs were prepared and read in by the RIM loader.

```
7756/6032      clear keyboard flag and accumulator
7757/6031      skip on keyboard flag set
7760/5357      jump .-1
7761/6036      read keyboard
7762/7106      clear link and rotate accumulator-link twice left
7763/7006      rotate accumulator-link twice left
7764/7510      skip on positive accumulator (bit 0 = 0)
7765/5357             else jump to address 7757
7766/7006      rotate accumulator-link twice left
7767/6031      skip on keyboard flag set
7770/5367      jump .-1
7771/6034      or keyboard buffer with accumulator
7772/7420      skip on non-zero link
7773/3776      deposit AC to effective address contained at 7776; clear AC
7774/3376      deposit AC to 7776
7775/5356      jump to 7756
```

Fig. 24. RIM Loader – see *Introduction to Programming – PDP-8 Handbook Series* p. 5-5

The format for RIM-loadable code consisted of address/contents pairs of octal numbers with the address and corresponding contents each split over two rows of paper tape. Only six tracks of the paper tape were needed for each half integer. To indicate an address, bit 7 was set to one for the upper half of the address. Thus address/contents pair 0200/7300 would be encoded as the four octal values

```
102     <- bit 7 is 1 : address
000
073     <- bit 7 is 0 : contents
000
```

The RIM loader combined pairs of numbers by reading the first six bits, shifting them to the left then literally Aor-ing@ in the second six bits. An address was deposited to location 7776o (bit 7 of the first number read was a 1); an instruction it was deposited to the location *pointed to* by location 7776o (the skip on non-zero link at address 7772o determined which). This continued until there was no more input to read. The RIM loader was synchronized by prefixing a number of 377o's to the tape before the code - the skip on positive accumulator at 7764o synchronized the program with the tape.

To use the RIM loader to load a program, first enter the RIM code into PDP-8 memory. (To make this more interesting use the **Switch Register**, **Load Addr**, **Deposit** and **Examine** switches to enter the RIM loader from the console!) Next create a text file in Abinary@ format containing the code you want to load (lines of three digit octal values between 000 and 377) making sure to insert lines of 377's at the beginning and end of the file (for synchronization). From the Run PDP-8 Program screen activate the paper tape reader by selecting **Reader.** When prompted enter the file name and select binary mode. To start the RIM loader enter starting address 7756o into the Program Counter register using the **Switch Register** and **Load Addr** buttons and select the **Go** button. The loader program should read the binary file into memory. When you think the file is read, use **Ctrt/C** to stop the program (equivalent to hitting the Stop button).

10.6 PDP-8 Interrupt System

Device 0 for opcode 6 was the PDP-8 interrupt system. When the interrupt system is enabled an external interrupt immediately disables the interrupt system and executes an automatic JMS 0 instruction. There are a number of opcode 6 interrupt instructions but only three are implemented by the PDP-8 Emulator Program.

Mnemonic	Opcode	Full Name : Description
SKON	6000	Skip if interrupts enabled and disable interrupts
ION	6001	Execute the next instruction then enable interrupts
IOF	6002	Disable interrupts

Table 7. PDP-8 Interrupt Instructions

I/O can be done more efficiently using interrupts. If the interrupt system is enabled, whenever an I/O device needed to be serviced (for example, a key is press on the keyboard or the printer is ready to print another character), an interrupt will be generated. This causes an automatic subroutine jump to location 0 (JMS 0) where an interrupt handler can Apoll@ the different devices to determine which device generated the interrupt..

The following PDP-8 program demonstrates how the interrupt system works. The main program starting at location 0200 endlessly prints out the alphabet in lower case. Whenever the user presses a key, an interrupt is generated. The automatic **jms 0** instruction branches control to location 0001 where a **jms i IntH** instruction branches control to the interrupt handler at location 5000. The handler saves the contents of the accumulator checks to see if the character read from the keyboard is a carriage return (and if so halts the program), sounds the "bell", outputs a line feed character, restores the accumulator then returns control to address 0002 where the **ion** instruction enables the interrupt system *following* the execution of the return from interrupt instruction (**jmp i 0**) at address 0003. Note that only the **ion** instruction is used.

```
/
/ Interrupt Routine
/
*0000
        0               / Return Address for Interrupt
        jms i IntH      / Jump to Handler
        ion             / Turn on Interrupt System
        jmp i 0         / Return from Interrupt
IntH,   XIntH           / Address of Interrupt Handler
/
/ Basic I/O Routine Vectors - Page 0
/
*0050
GetChar,  XGetChar
Type,     XType
CRLF,     XCRLF
PrtStr,   XPrtStr
/
/ Code Segment - Page 1
/
*0200
Main,   cla cll         / Clear AC and Link
        ksf             / Wake up keyboard
        tls             / Wake up printer
        ion             / Turn on Interrupt System
        cla cll
        tad A           / Get 'a'
        dca CH
Loop,   cla cll
        tad CH
        jms i Type      / Display letter
        cla cll
        tad CH
        iac
        tad MZ          / Check if 'z'
        sna
        jmp Reset       / If 'z' reset sequence
```

```
        isz CH          / Else increment ASCII code
        jmp Loop
Reset,  tad A           / Get 'a'
        dca CH
        jms i CRLF      / Do CR-LF
        jmp Loop
A,      'a'
CH,     0
MZ,     -123d           / minus ASCII 'z'
/
/ Interrupt Handler
/
*5000
XIntH,  0               / Interrupt Handler
        dca XHold       / Save AC
        krb             / Get character
        tad XCR         / Is it a CR?
        sna             /   No  - continue
        jmp XHalt       /   Yes - halt
        cla cll
        tad Beep        / Get Bell
        tsf             /
        jmp .-1
        tls             / Sound It
        cla cll
        tad LF          / Get Line Feed
        tsf
        jmp .-1
        tls             / Display It
        cla cll
        tad XHold       / Restore AC
        jmp i XIntH     / Return
Xhalt   hlt             / Halt Program
        jmp .-4         / Restart ?
XHold,  0
XCR,    -13d
Beep,   07
LF,     10d
/
/ Basic I/O Routines - Page 30
/
*7400
...   see BIOS code
$Main
```

Fig. 25. PDP-8 interrupt demo program

Note that the interrupt handler does itsown I/O. It does NOT make any calls to the simple character I/O routines on page 30. To do so would cause problems since if an interrupt occurred during the execution of one of these I/O routines, the interrupt handler would overwrite the original return address.

In the current implementation of the PDP-8 Emulator Program an interrupt can only be generated by hitting a key.

11. ADDITIONAL RESOURCES

Because of the commercial success of the PDP-8, it being the first widely sold mini-computer, references to the PDP-8 often occur in textbooks on computer organization (see Kraft and Toy, Stallings or Tanenbaum). An excellent source for information on the PDP-8 is Bell and Newell's book *Computer Structures: Reading and Examples*, which has a very nice article on the PDP-8 including a very nice ISP (instruction set processor) description. Another source of information is Bell, Mudge and McNamara's *Computer Engineering: A DEC View of Hardware Systems Design*., which deals more with the hardware aspect of the PDP-8 within the context of other DEC computers. A more current text is Blaauw and Brook's *Computer Architecture Concepts and Evolution*, which also contains a detailed architectural description of the PDP-8 set in the context of other computer architectures. And I would be negligent if I failed to mention the availability of numerous web-based resources for the PDP-8; a simple search on the word PDP-8 will bring up numerous sites with links to color photos of PDP-8's. A good web site to start at is Doug Jones' DEC PDP-8 Index (http://www.cs.uiowa.edu/~jones/pdp8/) which contains among other things a very nice link to a Programmer's Reference Manual for the PDP-8 (http://www.cs.uiowa.edu/~jones/pdp8/man/). PDP-8 hardware and software manuals, when available, are a major source for understanding the inner workings of the PDP-8. In particular the two texts from the PDP-8 Handbook series listed under References provided a wealth of programming information and techniques for the PDP-8. Most of the programming examples presented in this paper were derived from examples in these manuals. I also have a custom published text, *The PDP-8 Emulator User's Manual*, which I use in teaching computer organization, electronic copies of which are available from me.

12. SUMMARY AND OBSERVATIONS

Because the PDP-8 Emulator Program captures the simple and elegant architecture of the PDP-8, it makes an ideal introduction to computer organization. With a not-very-steep learning curve and an easily understood architecture, students can quickly learn to write PDP-8 programs. In my own computer organization course I have three labs and four programming assignments of increasing difficulty which make use of the PDP-8 Emulator Program (including one exercise where students have to write machine code just to give them the experience of having done it at least once). However, I do not use the PDP-8 as the main architecture to study; currently the Intel 80x86 architecture holds that role. But using the PDP-8, I am able to compare and contrast the two architectures demonstrating design trade-offs and showing that there more than one way to do things architecturally.

The PDP-8 is a good example of how a simple instruction set can execute complex programs and I point out that there is something "RISC-y" about the PDP-8. I also like the fact that the simple opcode 6 instructions permit the user to write all of his/her own I/O routines – something that cannot be easily (or realistically) done with today's architectures. Finally the PDP-8 is a *real* architecture (warts and all) – not a toy architecture. Why create a toy architecture to teach computer organization when there is a real one readily available?

ACKNOWLEDGEMENTS

Thanks to Ken Irwin at Wittenberg for converting my files to .pdf format and Bill Yurcik at Illinois Wesleyan for his helpful and greatly appreciated editorial suggestions.

REFERENCES

_____ 1973. *PDP-8/e, PDP-8/m & PDP-8/f Small Computer Handbook - PDP-8 Handbook Series* . Digital Equipment Corporation, Maynard, MA.

_____ 1975. *Introduction to Programming - PDP-8 Handbook Series*. Digital Equipment Corporation, Maynard, MA.

BELL, C. GORDON, AND NEWELL, ALLEN 1971. *Computer Structures: Reading and Examples*. McGraw-Hill, New York, NY.

BELL, C. GORDON, MUDGE, J. CRAIG, AND McNAMARA, JOHN, E. 1978. *Computer Engineering A DEC View of Hardware Systems Design*. Digital Press, Bedford, MA.

BLAAUW, G.A. AND BROOKS, F.P. 1997. *Computer Architecture Concepts and Evolution*. Addison-Wesley Longman, Reading, MA.

KRAFT, G.D., AND TOY, W.N. 1979. *Mini/Microcomputer Hardware Design*. Prentice-Hall, Englewood Cliffs, NJ.

SHELBURNE, B. 2001. *A PDP-8 Emulator User's Manual*, custom published, Wittenberg University, Springfield, OH.

STALLINGS, WILLIAM 2000. *Computer Organization and Architecture 5th ed.*.Prentice Hall, Upper Saddle River, NJ.

TANENBAUM, ANDREW S. 1999. *Structured Computer Organization 4th ed.* .Prentice Hall, Upper Saddle River, NJ.

Received: November 2001; accepted: February 2002.